NetLogger

Reference Manual
version 3.3.0, updated 06-September-2005

Dan Gunter, Brian Tierney

This is the manual for NetLogger version 3.3.0

For further information about this notice, contact Dan Gunter, dkgunter@lbl.gov.

Table of Contents

1 Overview of NetLogger

1.1 What is NetLogger?

Anyone who has ever tried to debug or do performance analysis of complex distributed applications knows that it can be a very difficult task. Problems may be in many various software components, hardware components, networks, the OS, etc.

NetLogger is designed to make this easier. NetLogger is both a methodology for analyzing distributed systems, and a set of tools to help implement the methodology. In fact, you can use the NetLogger methodology without using any of the LBNL provided tools.

The NetLogger methodology is really quite simple. It consists of the following:

1. All components must be instrumented to produce monitoring These components include application software, middleware, operating system, and networks. The more components that are instrumented the better.
2. All monitoring events must use a common format and common set of attributes. Monitoring events most also all contain a precision timestamp which is in a single timezone (GMT) and globally synchronized via a clock synchronization method such as NTP.
3. Log all of the following events: Entering and exiting any program or software component, and begin/end of all IO (disk and network).
4. Collect all log data in a central location
5. Use event correlation and visualization tools to analyze the monitoring event logs.

The open source NetLogger Toolkit is a set of tools to make it easier to implement this methodology.

1.2 NetLogger features and benefits

- Lightweight
- Efficient
- Easy to integrate
- Flexible

1.3 A brief history of NetLogger

NetLogger has been in existence, in one form or another, since 1994. Since that time it has been rewritten and renamed, so that the body of software now labeled NetLogger has little or no relation to the software distributed in the early years of research and development.

Until October 2004, the current version of NetLogger was called NetLogger 'Lite'. This was in contrast to the previously developed toolkit known as NetLogger, which was then renamed NetLogger 'Full'.

The 'Full' NetLogger instrumentation library is very powerful and contains a number of features such as backup destinations, auto-reconnect, remote activation, a binary format, etc. Experience showed that most of these features were not needed for the most common use-cases, so we developed a new, lightweight, easy to use version of the NetLogger instrumentation library called, initially, NetLogger-lite. This is the version of the toolkit documented here.

NetLogger 'Full' is still available (tho unsupported), and available at http://dsd.lbl.gov/NetLogger/full/. The very fast binary format, described in our HPDC 2002 paper, is only supported by the full library, so programs which generate more than 1000 events/second will still want to use the full library.

1.4 Getting more Information

- **Quick-start guide**

  http://dsd.lbl.gov/NetLogger/NetLogger-quick-start/

- **FAQ**

  Several Frequently Asked Questions are outlined in the NetLogger FAQ. Do you have a question that's not in the FAQ, send it to us and we'll add it.

- **Tutorial**

  You can view or download the NetLogger tutorial slides from: http://dsd.lbl.gov/NetLogger//NetLogger-Tutorial.ppt

- **E-mail list**

  The email list is in the process of being created. Please email dkgunter@lbl.gov if you want to be notified when it exists.

- **Website**

  The official NetLogger website is located at:

  http://dsd.lbl.gov/NetLogger/

  It contains all of the above documentation and more.

2 Requirements

2.1 Hardware Requirements

NetLogger requires no custom hardware, and runs on low-cost commodity PC-style system.

Item	Requirements
Memory	No particular requirement
Hard disk	Enough space for the log files

2.2 Software Requirements

The following software is required to use NetLogger. The Java specific software is only needed if you are using the Java client API.

Item	Requirements
Operating System	NetLogger uses Python for most of its utilities and tools. Although in theory the Python code will run equally well on Unix, Windows, and MacOS, the software has only been extensively tested on Unix.
Other Software	Python version 2.3 or higher (http://www.python.org) Ant (http://ant.apache.org/index.html) Log4J (http://logging.apache.org/log4j) Commons (http://jakarta.apache.org/commons/logging/) JUnit (http://www.junit.org/)

3 Installation and Configuration

3.1 Download

Download NetLogger from: http://dsd.lbl.gov/NetLogger/

3.2 Install

3.2.1 Installation overview

The directory layout NetLogger is arranged by language. The Section 3.2.2 [Basic install], page 4 installation process uses GNU make (GNU make is available from http://wwww.gnu.org) to visit these directories and invoke the appropriate build tool. In the Section 3.2.3 [Advanced install], page 5 installation process, you do this part yourself.

In either case, you need to pick an installation *prefix*, which is the parent directory for script installation, and where Python and C libraries will be copied. This directory can be system-wide, or located in a user's home directory. A typical system-wide value is `/usr/local`, and a typical user directory is `$HOME/local`. Whatever value is picked, it will be referred to as `NLHOME` in the examples that follow.

Since almost all the tools are written in Python, and thus the installation of the Python language directory is necessary for most uses of NetLogger.

3.2.1.1 Installation order

If you want to use the C library, either directly or as a SWIG-wrapped (http://www.swig.org) library to speed up the Python instrumentation, you should build and install the C directory first. After that, the order of installed modules is irrelevant.

3.2.2 Basic build/install

1. Unpack the Section 3.1 [Download], page 4 file with Unix 'tar', then change to the unpacked directory.

```
tar xzvf NetLogger-3.3.0.tar.gz
cd NetLogger-3.3.0/
```

2. Unpack the subdirectories:

```
make unpack
```

3. Set the environment variable NLHOME

```
For sh, bash, zsh, etc.:
      NLHOME=/path/to/home ; export NLHOME
For csh, tcsh, etc.:
      setenv NLHOME /path/to/home
```

4. Build

```
make
```

5. If that succeeds, install:

```
make install
```

Note: If you change NLHOME after you build, you will need to run 'make' again before the C API will install to the new directory.

3.2.3 Advanced build/install

You can build language-by-language by specifying the language as the make target, e.g. to build 'C': make c

If you need more control over the build process, cd into the appropriate subdir and build there. Note that to properly build the Python/C wrappers, you should build the C *before* the Python directory.

3.2.4 Links to more information

GNU build tools

GNU make: http://www.gnu.org
Autoconf: http://www.gnu.org/software/autoconf/autoconf.html
Automake: http://www.gnu.org/software/automake/automake.html
Libtool: http://www.gnu.org/software/libtool/

Python Distutils module

http://www.python.org/doc/current/inst/inst.html

Apache Ant build tool

http://ant.apache.org/

Simplified Wrapper Interface Generator (SWIG)

http://www.swig.org

3.3 Configuration

After you are done building and installing NetLogger, you need to set a couple of environment variables to put the tools in your PATH.

PATH

Add NLHOME/bin to your PATH

```
For sh, bash, zsh, etc.:
      PATH=${PATH}:${NLHOME}/bin
      export PATH
For csh, tcsh, etc.:
      setenv PATH ${PATH}:${NLHOME}/bin
```

PYTHONPATH

If you installed the Python library system-wide (i.e. NLHOME was the parent directory of lib/python<version> for your default version of Python) then you don't need to set PYTHONPATH. Otherwise, add this to your startup scripts:

```
For sh, bash, zsh, etc.:
      PYTHONPATH=${NLHOME}/lib/python<version>/site-packages
      export PYTHONPATH
```

```
For csh, tcsh, etc.:
      setenv PYTHONPATH ${NLHOME}/lib/python<version>/site-packages
```

Java: CLASSPATH

Copy java/lib/netlogger.jar to an appropriate location and add it to your CLASSPATH. For example, if it is in ${NLHOME}/lib/java:

```
For sh, bash, zsh, etc.:
      CLASSPATH=${NLHOME}/lib/java/netlogger.jar:${CLASSPATH}
      export PYTHONPATH
For csh, tcsh, etc.:
      setenv CLASSPATH ${NLHOME}/lib/java/netlogger.jar:${CLASSPATH}
```

4 Using NetLogger

4.1 Concepts

4.1.1 Distributed logging

When monitoring a distributed system, it is important to be able to collect the log data remotely and automatically. Manually logging into remote systems and transferring log files is not a scalable procedure, and sometimes security restrictions may even make it impossible.

NetLogger has been designed from the start to include the ability to transfer data over the network to a "central" location. It is still possible to use NetLogger to transfer each log message directly over a TCP socket, but we no longer recommend this practice. Reliability and availability of wide-area networks is too variable to depend on them to transfer the requested data without loss or blocking the caller.

Instead, we recommend a "store and forward" approach: writing to local storage (e.g. local disk) and then, with a separate process, forwarding the information to the network. This has the advantage of putting the burden of dealing with network failures in a single external component, the forwarder, rather than making each instrumented program deal with it separately. Specifically, we recommend that users follow these steps:

- Write logs to local disk: The most consistently available medium that has sufficient performance and stability for logging is local disk, for example a file in /tmp.
- Run a separate program to send these logs over the network: By running a separate program, you can re-use its functionality to forward system-level statistics such as CPU and network utilization.
- Expect network failures: Failure to write to the network should never be fatal to the forwarder.
- Provide a means for cleaning up: The program can also handle cleanup of old or runaway log files (something that becomes very important as the application scales).

The NetLogger tools that perform this job are the forwarder, *nlforward*, and the network server *netlogd* (for usage details, see Section 4.4.1 [nlforward], page 18, and Section 4.4.2 [netlogd], page 20).

4.1.2 Versioning and Compatibility

Output formats

The NetLogger 3.0 release, and previous releases called "nllite", introduced a new ASCII format that is a slight variation on the ASCII format called *ULM* that was used by previous NetLogger releases. The binary format is not produced, and for the most part not consumed, by the NetLogger 3.0 or later. For details on this format see the next section.

4.1.3 Output format

Each NetLogger message is called an *event* or event "record". The event is subdivided into *fields*. Each field has three parts, in this order: the **typecode**, the **key**, and the **value**. Here is an example record:

```
t DATE: 2004-04-15T21:36:01.425059
s LVL: Info
s HOST: 131.243.2.143
s TGT: app
s EVNT: program.end
```

Each field in the event ends with a single newline ('\n') character. The last field in each event is a blank separator line, i.e. just a newline.

Typecode

The typecodes tell how to interpret the value.

'`d`' Double. Double-precision floating-point number (64-bit IEEE float)

'`i`' Integer. Signed integer from -2147483648 to 2147483648 (32-bit integer)

'`l`' Long integer. Signed integer from -9223372036854775808 to 9223372036854775808 (64-bit integer)

'`s`' String. Up to 4096 characters. Only printable letters are allowed, and the only whitespace allowed is a space or tab.

'`t`' Timestamp. One of the legal variations of the ISO8601 date format standard: YYYY-MM-DDThh:mm:ss.<fractional seconds>. The fractional seconds may be up to 9 digits (nanoseconds).

Key

The key is a string, with the same restrictions for string values (typecode 's', above).

Value

The value may be up to 4096 bytes long. Its interpretation depends on the typecode.

4.1.4 Output URL

URL format

The syntax of the URLs is indicated with a simple grammar. Square brackets=[optional]. Square brackets with a '*' means [0 or more]*.

File url Syntax: [file:][/][path]

Examples:

- Log to a file in /tmp: `/tmp/logfile`

TCP url Syntax: x-netlog://host[:port]

Examples:

- Log to localhost at port 14830: `x-netlog://localhost`
- Log to remote host: `x-netlog://remote.host:1143?`

UDP url (Python,Perl,C)
Syntax: x-netlog-udp://host[:port]

Examples:

- Log to localhost at port 14830: `x-netlog-udp://localhost`
- Log to remote host: `x-netlog-udp://remote.host:1143?`

Syslog url (C only)
Syntax: x-syslog:

Examples:

- Log to syslog: `x-syslog:`

4.1.5 Naming conventions

Standard fields

The following "standard" fields are always present in a NetLogger event:

'`DATE`' The time that this event occurred.

'`LVL`' The logging level of this event. This is one of: '`Fatal`', '`Error`', '`Warn`', '`Info`', '`Debug`', '`Debug1`', '`Debug2`', '`Debug3`', or '`User`'. The default level is '`Info`'.

'`EVNT`' This field is the name of the event. By convention (again, based on the DAMED WG document), this name is hierarchical with the most detailed information last and camel-case starting with a lowercase letter. But no component enforces this convention.

4.1.6 Environment Variables

XXX not done

explain the following ENV variables here

NETLOGGER_DEST

NETLOGGER_ON

NL_GID

4.2 Instrumentation API

4.2.1 C

Overview

To use the C API, follow this pattern:

- Create logging 'module'(s): `NL_logger_module()`
- Write messages to module(s): `NL_write()`, `NL_debug()`, `NL_info()`, `NL_warn()`, `NL_error()`, ..etc.
- Cleanup: `NL_logger_del()`

In the NetLogger C API, a **module** is a user-assigned name for all the parameters associated with a logging output – log "type", destination URL, and logging level. Every time you write a message, you specify which module to write it to. So, for example, you may specify one module for messages to the user on '`stdout`', and another one for detailed debugging to a file in '`/tmp`':

```
NL_logger_module( "user", "-" /* stdout */, NL_LVL_INFO, NL_TGT_X );
NL_logger_module( "debug", "/tmp/log", NL_LVL_DEBUG, NL_TGT_DBG, "" );
```

Example code

File	Description
logger_example.c (HTML) (source)	Sample high-level C API usage. This is the API that most people would be expected to use.
lowlevel_example.c (HTML) (source)	Sample 'low-level' C API usage, illustrating two things: (1) basic usage, and (2) how to use nl_mask() to select which sets of events are written at any given time.

Environment variables

If you provide a `NULL` url to NL_logger_module(), then NetLogger will look for the enviroment variable `NETLOGGER_DEST` for the URL. If NETLOGGER_DEST is not set, events will do to `stderr`.

For example, if you have this code:

```
#include "nl_log.h"
main() {
   NL_logger_module("dbg", NULL, NL_LVL_INFO, NL_TYPE_DBG, "");
   NL_info("dbg","Hello","to_whom=s","World");
}
```

Then NetLogger events will be sent to the URL specified in `NETLOGGER_DEST`.

Note: only the C API looks at this environment variable. The Python, Java, and Perl APIs do not.

Function summary

NL_logger_module()

```
NL_result_t
NL_logger_module( const char *module,
                  const char *url,
```

```
NL_level_t level,
NL_tgttype_t ttype,
... );
```

Parameters

The *module* name is a free-form string, but please remember that this is a key in a (hidden, global) hash-table, so long descriptive names will hurt performance.

The *url* is a standard netlogger URL. With no scheme or "file://" it is a filename, with "x-netlog://<host>[:<port>]" it is a TCP destination, and with "x-syslog://" it uses syslog. Two special filenames are defined, "-" for stdout and "&" for stderr. If the URL is NULL, the value of the environment *NL_DEST* is used, otherwise '`stderr`'.

The *level* is an integer. Higher numbers indicate more detailed logging, or looked at another way lower numbers are more 'critical' messages. The header file defines these symbolic names:

NL_LVL_NOLOG = 0
: Log nothing

NL_LVL_FATAL = 1
: Fatal error (program termination expected)

NL_LVL_ERROR = 2
: Error

NL_LVL_WARN = 3
: Warning

NL_LVL_INFO = 4
: Informational (default level)

NL_LVL_DEBUG = 5
: Debugging message

NL_LVL_DEBUG1 = 6
: More detailed debugging

NL_LVL_DEBUG2 = 7
: Even more detailed debugging

NL_LVL_DEBUG3 = 8
: Even more detailed debugging yet

NL_LVL_USER = 9
: This level and all others after it are not reserved

The *ttype*, or target type, is an integer constant. Different target types will write different information, by default, in each message. The header file defines symbolic names:

NL_TYPE_APP
: "app". Application instrumentation

NL_TYPE_DBG
: "dbg". Application debugging

NL_TYPE_PATH
: "path". Network path monitoring

NL_TYPE_NODE
: "node". Host, etc. monitoring

NL_TYPE_X
: "any". None of the above..

The ... (variable arguments) depend on the target type. For '`APP`', '`DBG`', and '`NODE`', the user should add the host IP address or `""` to make NetLogger determine it automatically (via DNS). For the other target types, no argument is expected.

Return value

If the module was created, then NL_OK is returned. Otherwise NL_ERROR is returned, and an error message should have been written to the screen (stderr).

NL_write()

The various logging functions are named for the logging level they use. The generic function is `NL_write()`, which unlike the others takes the level as its first argument.

- NL_write(level, module, event, fmt, args...) Write any level message
- NL_debug3(module, event, fmt, args...) Debug level 3 message.
- NL_debug2(module, event, fmt, args...) Debug level 2 message.
- NL_debug1(module, event, fmt, args...) Debug level 1 message.
- NL_debug(module, event, fmt, args...) Debug (level 0) message.
- NL_info(module, event, fmt, args...) Informational message.
- NL_warn(module, event, fmt, args...) Warning message.
- NL_error(module, event, fmt, args...) Error message.
- NL_fatal(module, event, fmt, args...) Fatal error message.

Parameters

The *module* is the name given to the module when it was created with `NL_logger_module(()`.

The *event* is the name of the event.

The format string (*fmt*) describes the format of the variable arguments to follow, with the exception of the "special" arguments required for certain "target types", as explained below.

The syntax for *fmt* is `<field>=<type-code> space <field>=<type-code> ..etc..`, where <field> is the name of "key" of the key/value pair, and "type-code" is a one-letter code indicating the datatype. The type codes are enumerated in Section 4.1.3 [Output format], page 7. To improve performance, you can use a `":"` instead of a `"="` for values that will never change (a.k.a. constants) from their initial value.

For the *path* target type, the first two arguments are always interpreted as '`"SRC=s DST=s"`', i.e. the source and destination endpoints. Therefore, when using this target type, if *fmt* was `"VAL=i"` then you would have three (3) values: `NL_info("test","my.event", "VAL=i","12.13.14.15","15.14.13.12","76");`

NL_flush

There is no internal buffering of NetLogger writes. However, the operating system will buffer writes to socket or disk for improved efficiency. To force synchronization of data to the physical medium after every write, set the global variable NL_flush to '1'. For example:

```
void my_function( char *msg ) {
    NL_flush = 1;
    NL_debug("main","hello","MSG=s",msg);
}
```

Activation

The C API also includes the ability to turn on/off instrumentation in a running program. This is done using an *activation trigger file*.

XXX: explain activation file format here.

4.2.2 Java

Instrumentation API [Javadoc]

File	Description
LogMessage.java (HTML) (source)	A NetLogger message is structured like a dictionary, in other words a set of strings called "field names", each mapped to a value which is either a string, integer, or floating-point number, called "field values". At creation time, the message event name is set, as well as the "target type", i.e. kind of logging being performed. Further name and value pairs are added into the record with add() methods which, by virtue of returning the newly modified LogMessage instance, can be chained together. For example: LogMessage message = new LogMessage("my.event","app").add("my int",3).add("my float",4.0); The user can set the timestamp to something other than the result of system.getCurrentTimeMillis() by calling setTimeStamp(). To format the message as a NetLogger record, call toString(). @@author Dan Gunter dkgunter@@lbl.gov @@author Wolfgang Hoschek whoschek@@lbl.gov @@version $Revision: 1.4 $

Examples

File	Description
NLSample.java (HTML) (source)	Sample use of NL formatter with Java logging facility. Requires commons and log4j libraries to run. Run it along the following lines: java -cp build/classes:lib/log4j.jar:lib/commons-logging.jar -D

4.2.3 Python

Instrumentation API [epydoc documentation]

File	Description
__init__.py (HTML) (source)	
autolog.py (HTML) (source)	At run-time, automatically instrument a Python module, class, or (list of) functions. A NetLogger write call is inserted before and after the function invocation. The logging object, function called to write a message, event name suffixes, etc. are all configurable.
fakelogging.py (HTML) (source)	Imitiate API of Python logging module, for clients running in Python < 2.3.x Use this formula: try: import logging # preferred! except ImportError: import gov.lbl.dsd.netlogger.fakelogging as logging log = logging.getLogger('my_logger')

nllite.py (HTML) (source)	NetLogger instrumentation API for Python Read and write NetLogger log messages. Most users of this API will use the LogOutputStream, which is like a 'Logger' object in the Python logging API. The LogInputStream will perform the converse operation: read and parse a stream of messages. If available, native C functions will be used for serialization and deserialization. Utility functions include a 'Date' class that automatically handles conversion between string representations and floating-point seconds since the epoch, functions to get and set the Grid Job ID, and low-level parse() and format() functions.
nllogging.py (HTML) (source)	NetLogger adaptation layer for Python's 'logging' module. Provides wrappers for logging.Handler classes, which allow them to write NetLogger messages instead of a simple string message. The logging call is modified to indicate that structure, so that instead of a string input and list of arguments, the log call takes an event name and dictionary of name/value pairs.
socketserver.py (HTML) (source)	An embeddable NetLogger socket-server. All the user needs to do is provide a port and a callback function that will receive the parsed NetLogger nllite.Record objects. The implementation is based on the standard 'asyncore' module.

Examples

File	Description
autolog_example.py (HTML) (source)	Example use of autolog API.

csv_example.py
(HTML) (source)

```
Example generation of CSV files from NetLogger logs.

Usage:
python csv_example.py < csv_example.log
```

nllite_example.py
(HTML) (source)

```
Example use of the "low-level" NetLogger Python API.
This API is much more efficient (by a factor of 10-30) than
the Python 2.3 "logging" module interface.
```

nllogging_example.py
(HTML) (source)

```
Use of NetLogger through the standard Python (2.3 and up) 'logging' module
```

4.2.4 Perl

Instrumentation API [POD documentation]

File	Description
NetLogger.pm (HTML) (source)	*NAME* NetLogger - NetLogger logging API. *SYNOPSIS* Provides formatting and timestamping of messages, writing to file and socket. *DESCRIPTION* Provides formatting and timestamping of messages, writing to file and socket.

Examples

File	Description
sample.pl (HTML) (source)	
perf.pl (HTML) (source)	

4.3 Configuration file formats

As of version `3.2.15`, all XML configuration files can either be in XML or use an easier-to-type format called 'Compact XML', or CXML. Examples in this manual will be given in this format, as well as XML. The syntax of XML is already explained in great detail by others. The syntax of CXML is explained in detail in this section.

Note that CXML is not supposed to be some sort of replacement for XML. It's just meant to help out in manual editing of XML configuration files.

Programs that use XML/CXML:

- Section 4.4.3 [nldemux], page 21
- Section 4.5.3 [nlfindmissing], page 37

4.3.1 CXML Description

- Header: The first line must be: #CXML#
- Line-orientation: Each element begins on a new line
- Comments: Lines after the first, whose first non-whitespace character is a pound (#), are comments.
- Indentation: Indentation indicates nesting.
- Tags: A trailing colon (:) indicates the end of the tag
- Attributes: Name/value attributes on an internal node precede the colon, are enclosed in square brackets, ([]), name and value are separated by equals (=) and each pair is separated by a comma (,).
- Text: Text goes after the colon. It may contain whitespace, including newlines. Colons must be quoted with a backslash (\).
- Lists: A special format for text is a list of items. The format is a series of strings enclosed in square brackets ([]), separated by commas (,). Lists may contain newlines. Whitespace between the comma and start of next list item is ignored.

4.3.2 CXML Examples

Here is an example CXML file:

```
#CXML#
# This is a comment
Root:
   Sub-one: [ item one, item two ]
   Sub-two [color=blue,material=cotton]: descriptive text
   Sub-three: NOT only leaf nodes may have text.
       Sub-sub-one: Most deeply nested
```

And the equivalent XML, slightly cleaned up to be more readable:

```
<?xml version='1.0' ?>
<!--  This is a comment -->
<Root>
  <Sub-one>
    <item>item one</item>
    <item>item two </item>
  </Sub-one>
  <Sub-two color='blue' material='cotton'>descriptive text
  </Sub-two>
   <Sub-three>NOT only leaf nodes may have text.
      <Sub-sub-one>Most deeply nested</Sub-sub-one>
   </Sub-three>
</Root>
```

4.3.3 Converting from CXML to XML

The CXML parser is written in Python, and is part of the NetLogger Python library. In the source distribution, it can be found under `python/gov/lbl/dsd/netlogger/util/cxml.py`. If the installation Python directory is `NL_PY_HOME` this file can also be run as a simple conversion program that reads from *file.cml* and prints XML to standard output, as follows:

```
python $NL_PY_HOME/site-packages/gov/lbl/dsd/netlogger/util/cxml.py file.cml
```

Note that there is no tool to convert from XML to CXML. There is also no guarantee that arbitrary XML documents *can* be translated without loss of information to CXML! CXML is just meant to make manual entry of configuration files easier.

4.3.4 Tips on editing CXML

Any text editor which can edit Python code should do the indentation needed by CXML automatically. This works because the `:` used to indicate nesting in CXML is the same delimiter used to separate class and function declarations from their (nested) definition in Python. In addition, the square-bracket lists are similar enough to Python's list notation to allow automatic bracket-matching to work.

This has been tested and seems to work well in emacs. A full list of editors available for Python is kept at: Python editor wiki.

4.4 Tools

This section includes instructions for running several NetLogger tools, including tools to collect and forward log files, and tools to sort, merge, and extract data from the log files.

4.4.1 nlforward

Synopsis

```
Forward a directory of log files to a NetLogger URL.
```

Usage

```
usage: nlforward.py [options]  DIR/PATTERN URL [ URL .. ]

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -eSEC, --expire=SEC   how long, in seconds, before a file is considered
                        'expired' (default=3600 == 1 hour)
  -E, --erase           remove (erase) files from disk when they
                        expire(default=False)
  -F, --no-flush        do NOT flush after each record
  -pFILE, --persist=FILE
                        load/save state in file
  -r, --regexp          intepret PATTERN as a regular expression, instead of a
                        Unix 'glob' (default=False)
  -sSEC, --scan-interval=SEC
                        how often, in seconds, to scan DIR for new files
                        matching PATTERN (default=30)

  debug options:
    -q, --quiet         print nothing to stderr, overrides '-v'
    -v, --verbose       verbose mode (report throughput)
    -d, --debug         synonym for -v/--verbose
```

Detailed description

nlforward scans a directory, and sends every file matching the specified pattern to a NetLogger output destination (given as a URL). For explanation of why this tool is important, See Section 4.1.1 [Distributed logging], page 7.

Scanning entire directories, rather than sending a single file, is essential if you have multiple independent processes in your instrumented application. For file consistency, only one process may be writing to a given file. So, each process writes to its own file, but all files are in the same directory, and the whole directory gets forwarded at once. Nothing could be easier!

In addition to looking at file modification times and marking files as "expired", nlforward is also able to recognize when a file is overwritten, i.e. when its creation time changes. This information is also remembered in the persistent state file (`-p` option), so that files which are replaced between the time the forwarder is stopped and restarted are considered "new".

This feature is useful in a variety of contexts, but one common scenario is that the logging system is "rolling over" a file to a backup file and then re-using that filename for new data. Assuming that you have configured *nlforward* so that only the main file (not the rolled-over copies) matches the pattern of files to be forwarded, then each log record will be forwarded correctly, and only once.

Example

```
nlforward  -q -E -e 600 -s 30 ./'*.log' -p state.dat x-netlog://some.host:14380
```

Scan the current directory for files ending in ".log" and forward them to the netlogd on "some.host" port 14380. Files that have not been modified in the last 10 minutes are considered inactive (closed by the application) and will be erased once all their contents are forwarded. The current position in various files being forwarded is saved across runs in the file "state.dat".

4.4.2 netlogd

Synopsis

```
NetLogger socket server command-line program.

Features:

* listen on either a TCP or UDP socket.
* write to multiple output files.
* write in either NetLogger or older ULM format
```

Usage

```
usage: netlogd.py [options] [-h]

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -a, --action          Look for NL.ACTION field, act on 'flush' and 'close'
                        values
  -b, --fork            fork into the background after starting up
  -f, --flush           flush all outputs after each record
  -pPORT, --port=PORT   port number (default=14380)
  -rSIZE, --rollover=SIZE
                        roll over files at given file size (units allowed)
  -t, --timestamp       add a "time received" to each message
  -U, --udp             listen on a UDP instead of TCP socket

  Debug options:
    -q, --quiet         print nothing to stderr, overrides '-v'
    -v, --verbose       verbose mode (report throughput)
    -d, --debug         synonym for -v/--verbose
```

```
Output options:
  -oURL, --output=URL
                        Output URL, use 'nllite' format
  -uURL, --ulm-output=URL
                        Output file, use ULM format
  -xURL, --xml-output=URL
                        Output file, use XML format
```

Detailed description

netlogd is a TCP or UDP socket-server that can demultiplex any number of TCP streams, then re-send the resulting stream to one or more output destinations. The incoming format must be the standard NetLogger text format, but the output format can also be the older ULM format, or XML.

Typically, netlogd is configured to write its data to a single output file. But netlogd has been designed to be flexible. It can have multiple output destinations, each of which can be a file, syslog, or socket URL. Therefore, it could be used as a "repeater", i.e. it could both dump a local log file and forward the data on to second network server.

To handle the possibility of very large files, a simple rollover capability has been built in to netlogd. Once a file reaches a given size, a timestamp is appended to the filename and a new file is started.

Example

```
netlogd.py -p 14830 -o logfile -o x-netlog://other.host
```

Save all streams coming into TCP port 14830 to the file 'logfile', and also write a copy to the default port on "other.host".

UDP Example

```
netlogd.py --udp -p 14830 -o logfile -o x-netlog://other.host
```

Save all streams coming into UDP port 14830 to the file 'logfile', and also write a copy to the default port on "other.host".

4.4.3 nldemux

Synopsis

```
Split an input stream of NetLogger records into multiple
output streams, assigning records to file(s) based on their contents.
In a networking context, this might be called 'content-based routing'.
```

```
usage: nldemux.py [options] config-file < data-file

options:
      --version           show program's version number and exit
      -h, --help          show this help message and exit
      -d, --debug         Print debugging info
      -e, --stop-on-eof   Stop on EOF from input
      -v, --verbose       Synonym for -d/--debug
```

Usage

Detailed description

The *nldemux.py* tool allows input streams to be split across multiple files and directories, either of which can be rolled over in response to timeouts or netlogger log messages containing the special name/value pair ("NL.ACTION","close",type=string).

Typically, *nldemux.py* is used in conjunction with Section 4.4.2 [netlogd], page 20, to split and roll over log files collected from multiple distributed network senders. For examples, see the Usage section below.

Configuration

The behavior of *nldemux.py* is controlled by its configuration file. This file is in XML. Below the root element, whose name is ignored (examples use <root>), there are one or more <section> elements, containing information about which messages to accept and where to put them.

Configuration file format

```
#CXML#
root:
    # Each section defines a demultiplexing action.
    # Sections are named.
    section [name=NAME]:
        # Fields used by this section
        fields: [field,field,..]
        # How to split records into files. Field values
        # are shown as ${FIELD}.
        file [pattern=pattern-with-field-values]:
        # How to split records into directories. Field values
        # are shown as ${FIELD}.
        dir [pattern=pattern-with-field-values]:
        # Rollover, type can be 'file' to roll over into
        # a new file, or 'dir' to roll over entire directories
        # of files at once.
        rollover[type=file|dir]:
            # When nldemux gets a 'close' action, roll over immediately.
            onClose:
            # Extension pattern for rolled-over files
            ext[pattern=pattern-with-field-values]:
            # Timeout time, with units
            timeout: N days | N hours | N minutes | N seconds
    # Optional. If record matches no section, then it will match the
    # default section.
    default:
        # default section has just a simple file pattern
        file [pattern=file-name]:
```

Sections

There can be any number of sections. Each section provides information needed to choose the output path (dir + filename) from an input record. Each item of information in a section is called an 'option'. A given record may "match" zero or more sections in the configuration.

Default section

If a record matches no sections, and there is a section with the special tag <default>, the record will be written to the filename and directory specified in the default section.

Section Contents

A section contains one or more record field names, a file pattern and an optional directory pattern.

The file pattern specifies how to construct file names from the values of the input record fields. Any construct of the form "${FOO}" will be replaced with the input record's value for the field 'FOO'. All other characters are copied verbatim. Although this syntax looks like Unix shell variables, unlike in the shell the enclosing {} are required. If that last sentence made no sense to you, don't worry aout it.

The directory pattern, which is optional, follows the same rules for substitution.

Matching Records to Sections

An input record "matches" a section if all fields named in the section are also present in the record.

Note that all fields in a section do not need to be used in file or directory substitution. Nevertheless, their presence serves as a filter selecting input records that have those fields.

Flushing files

If a NetLogger record with the special name/value pair:

```
s NL.ACTION: flush
```

is received, then flush the matching output/section, where "matching" is defined as if the record were being written.

There is a special syntax used to flush all outputs in a section. The record sent should contain all fields used for determining the output file name (Note: *not* necessarily the same as the list of <field> elements), and all of them should be empty.

For example, if the configuration file contained the following section:

```
section [name=flushme]:
  fields: [Red,Blue]
  file [pattern=${RED}.log]:
```

Then, the following record would flush only the file "mauve.log":

```
t DATE: 2005-02-03T23:12:56.000000
s EVNT: anything
s LVL: INFO
s Red: mauve
s Blue: turquoise
```

and the following record would flush all outputs:

```
t DATE: 2005-02-03T23:12:56.000000
s EVNT: anything
s LVL: INFO
s Red: x
s Blue: turquoise
```

In essence, this facility combined with the Section 4.4.4 [nlwrite], page 26 utility, allows a simple form of RPC with *nldemux*.

Rolling over files

In order to support moving files out of the way, this program has a <rollover> configuration option. The suboptions are 'onClose', 'timeout', and 'ext'. 'onClose' is an optional empty element (i.e. a flag), which directs nldemux to roll over the output file in response to

the"NL.ACTION: close" name/value pair. 'timeout' directs timed rollover of the file, and accepts plain-english time periods separated by commas, such as "1 day, 4 hours". 'ext' is a pattern for the rolled-over filename extension. If the old file was "myfile", the new file is named "myfile.<ext>", where ext is by default the date in ISO8601 format, ie the special code ${__date__}.

The recognized pattern special codes are:

- ${__count__} rollover number (since program started)
- ${__sec__} number of whole seconds since epoch, UTC
- ${__date__} yyyy-mm-ddThh:mm:ss, UTC
- ${FIELD} last value of record field 'FIELD'

Sample configuration file (CXML, Section 4.3 [Configuration file formats], page 16)

```
#CXML#
root:
    section [name=app_logs]:
        fields: [RUN,USER,experiment]
        file [pattern=myapp${RUN}_${USER}.log]:
        dir [pattern=user-${USER}]:
        rollover[type=file]:
            onClose:
            ext[pattern=exp-${experiment}]:
    section [name=ganglia_logs]:
        file [pattern=ganglia.log]:
        # default 'dir' is '.'
        rollover [type=file]:
            onClose:
            timeout: 12 hours
    default:
        file [pattern=default.log]:
```

Sample configuration file (XML)

```
<?xml version='1.0'?>
<root>
  <section name='app_logs'>
    <fields>
      <item>RUN</item>
      <item>USER</item>
      <item>experiment</item>
    </fields>
    <file pattern='myapp${RUN}_${USER}.log'></file>
    <dir pattern='user-${USER}'></dir>
    <rollover type='file'>
      <onClose></onClose>
      <ext pattern='exp-${experiment}'></ext>
    </rollover>
  </section>
  <section name='ganglia_logs'>
    <file pattern='ganglia.log'>
      <!--  default 'dir' is '.' -->
    </file>
    <rollover type='file'>
      <onClose></onClose>
      <timeout>12 hours</timeout>
    </rollover>
  </section>
  <default>
    <file pattern='default.log'></file>
  </default>
</root>
```

Example

```
netlogd.py -o'-' | nldemux.py demux.cfg
```

Take the output stream from a NetLogger daemon and write it to files as described by "demux.cfg"

4.4.4 nlwrite

Synopsis

`Write a single message to a NetLogger output URL`

Usage

```
usage: nlwrite [ options ] URL [ name=value .. ]
    Special values:
    {n}            Message number
    {g}            Grid Job ID (will look in env, then make UUID)
    {h}            Host name

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -eEVENT, --event=EVENT
                        Event name (default=nlwrite.event)
  -F, --no-flush        Do NOT flush after every message
  -lLEVEL, --level=LEVEL
                        Level name ['FATAL', 'ERROR', 'WARN', 'INFO', 'DEBUG',
                        'DEBUG1', 'DEBUG2', 'DEBUG3'] (default=INFO)
  -nNUM, --num=NUM      Number of times to repeat the message
  -pPAUSE, --pause=PAUSE
                        Pause between each message in seconds (default=0)
```

Detailed description

nlwrite sends formatted NetLogger messages to any valid NetLogger URL. The event name, message level, and user-defined fields and values can all be specified. Any number of duplicate messages can be written to that destination.

This tool is useful for testing and debugging deployed NetLogger components, sort of like a "NetLogger ping". It also can be used for logging from shell scripts. For example, you could use it to instrument the start and end of a forked job. [nlwrite_examples], page 27.

Examples

- Write 1 message to a socket ('ping')

  ```
  nlwrite x-netlog://somehost.somedomain
  ```

- Write a start/end event for a forked job in a shell script Write to a file 'main.log' Include local host name, and Grid Job Identifier

  ```
  #!/bin/sh
  # Shell script
  echo "do some work"
  # set NL_GID environment variable so that
  # both nlwrites pick up on it:
  NL_GID=‘uuidgen‘; export NL_GID
  echo "fork child 'foo'"
  nlwrite -e foo.start main.log HOST={h} GID={g}
  foo
  nlwrite -e foo.end main.log HOST={h} GID={g}
  echo "foo is done, continue.."
  ```

```
#####

$ cat main.log
t DATE: 2004-10-08T20:55:17.237844
s LVL: INFO
s HOST: 131.243.2.143
s GID: a1d9c2a0-6af7-4148-be4e-b41f4661b311
s EVNT: foo.start

t DATE: 2004-10-08T20:55:17.327379
s LVL: INFO
s HOST: 131.243.2.143
s GID: a1d9c2a0-6af7-4148-be4e-b41f4661b311
s EVNT: foo.end
```

- Write, to stdout, 100 DEBUG messages. Put the message number in the field 'index'.

```
nlwrite -n 100 -e my.message -l DEBUG '-' index={n}

...

t DATE: 2004-10-08T21:03:26.051663
s LVL: DEBUG
i index: 99
s EVNT: my.message
```

4.4.5 nlconvert

Synopsis

```
Convert an 'nllite'-format file to ULM, or vice-versa.

The format of the input file is determined using an extremely
simple algorithm:

First 5 characters are 'DATE='?
Yes: ULM
No: NetLogger
```

Usage

```
usage: nlconvert.py [options] < logfile

options:
  --version   show program's version number and exit
```

```
-h, --help  show this help message and exit
```

Detailed description

Convert an ascii format file to ULM, or vice-versa.

Example

- ULM input file

  $ cat file1.ulm

  ```
  DATE=20040907154737.022505 HOST=127.0.0.1 PROG=unknown NL.EVNT=start_requestRead NL
  DATE=20040907154737.022505 HOST=127.0.0.1 PROG=unknown NL.EVNT=start_requestRead NL
  ```

- Convert to standard ASCII format

  $ cat file1.ulm | nlconvert

  ```
  s DATE: 2004-09-07T15:47:37.022505
  s HOST: 127.0.0.1
  s PROG: unknown
  s EVNT: start_requestRead
  s NL.ID: dWoAABnYPUF6UAAA
  s LVL: 0
  s int1: 1

  s DATE: 2004-09-07T15:47:37.022505
  s HOST: 127.0.0.1
  s PROG: unknown
  s EVNT: start_requestRead
  s NL.ID: dGoAABnYPUHbTwAA
  s LVL: 0
  s int1: 1
  ```

- Round-trip; convert back to ULM

  $ cat file1.ulm | nlconvert | nlconvert

  ```
  DATE=20040907154737.022505 HOST=127.0.0.1 PROG=unknown NL.EVNT=start_requestRead LV
  DATE=20040907154737.022505 HOST=127.0.0.1 PROG=unknown NL.EVNT=start_requestRead LV
  ```

4.4.6 nlgrep

Synopsis

Usage

```
usage: nlgrep.py [options] regexp file [ file ... ]

options:
  --version        show program's version number and exit
```

```
-h, --help       show this help message and exit
-e, --error      Print matches to stderr instead of stdout
-c, --count      Print count of matches instead of matches
-H N, --head= N  Print N records at head of file
-T N, --tail= N  Print N records at tail of file
-v, --reverse    Invert the sense of matching, to select non-matching
                 records
```

Detailed description

nlgrep (also callable as nlgrep.py) filters NetLogger records like the Unix command-line utility `grep`. Because the NetLogger ascii format spans multiple lines, 'grep' does easily not filter whole records.

Note that plain old 'grep -v' is still useful, however, for eliminating certain fields from *within* each record. For example, the script "grep -v 'EXTRA:' file" will eliminate all fields named 'EXTRA' from each record in which it occurs in a log file.

Examples

- Show records with event 'bar' (the input file is called 'foo.log').

  $ nlgrep 'EVNT: bar' foo.log
- Show records with event 'bar' or event 'baz' (the input file is called 'foo.log').

  $ nlgrep 'EVNT: bar|EVNT: baz' foo.log
- Show records with HOST in domain 131.243.2 (the input file is called 'foo.log').

  $ nlgrep 'HOST: 131\.243\.2' foo.log
- Count records with HOST in domain 131.243.2

  $ nlgrep –count 'HOST: 131\.243\.2' foo.log
- Count records with HOST *not* in domain 131.243.2

  $ nlgrep –reverse –count 'HOST: 131\.243\.2' foo.log
- Show first 10 records with HOST *not* in domain 131.243.2

  $ nlgrep -H 10 'HOST: 131\.243\.2' foo.log
- Show last 10 records with HOST *not* in domain 131.243.2

  $ nlgrep -T 10 'HOST: 131\.243\.2' foo.log

4.4.7 nlmerge

Synopsis

```
Merge multiple NetLogger log files (sorting by timestamp).

This will work on any combination of one or more files that needs
to be sorted by time. In other words, it is the way to sort a very
large file, as well as combine a large number of very small files.
```

```
The default output is 'nlsort.out' There is an option
to set the output file; use a dash ('-') to indicate standard output.
```

Usage

```
usage: nlmerge.py [ options ] file file [ file ... ]

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -bBATCH, --batch=BATCH
                        Number of records per temp file (default=1000)
  -DDIR, --dir=DIR      Directory for temporary files (default=/tmp)
  -oFILE, --output=FILE
                        Output file (default=nlsort.out)
  -tT1..T2, --timerange=T1..T2
                        Include only records between time T1..T2, inclusive.
                        Use two dots to separate the times, given in YYYY-MM-
                        DDThh:mm:ss.123456 format

  Debug options:
    -q, --quiet         print nothing to stderr, overrides '-v'
    -v, --verbose       verbose mode (report throughput)
    -d, --debug         synonym for -v/--verbose
```

Detailed description

nlmerge combines multiple logfiles into a single, time-ordered, file. It uses temporary disk files while merging, so the size of the merged files is limited by disk and not memory.

Note that you can use nlmerge as a file sorting tool if you give it only one input file.

Example

```
nlmerge foo1.log foo2.log foo3.log > merged.log
```

4.4.8 nldata

Synopsis

```
Transform NetLogger (text) -> Lifelines, Load/Scatter (text)
```

Usage

```
usage: nldata.py [options] config-file
```

```
options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -F, --format          print format info
  -iFILE, --input=FILE  Input file (default=stdin)
  -oFILE, --output=FILE
                        Output file (default=stdin)
  -v, --verbose         Verbose output to stderr (e.g., progress)
```

Detailed description

nldata.py transforms NetLogger log files into a comma-separated values (CSV) representation of "lifelines", "loadlines", and "points". For explanation of what those are, http://dsd.lbl.gov/NetLogger/nlv/.

It takes a configuration file describing which events belong together in lifelines, etc., and how to group them. Then it produces a simple output format.

Because they read from stdin and write to stdout by default, nldata and nldata_gp can be set up in a Unix pipeline to gnuplot itself. [nldata_gp_examples], page 34.

Configuration file format

```
title: My Graph
type: lifeline
events: a,b,c
labels: start,"",end
ids: GID
groups: HOST
# This is a comment.
# val: not needed for lifelines
% ( end of record )
type: point
events: d,e,f
groups: GID
val: VAL
%
# Draw state-transitions, etc.
# as a 'square waveform'
type: square
events: d
labels: state_one,state_two,state_three
val: state-value-field
```

Output format

ROW	Data
1	Title # Dataset title
2	t1,t2 # x-axis (time) range as 2 floats

3 label,label/ra,nge;label,label/range; ... # y-axis labels, ';' between groups / min,max of all values
4 label,label, ... # 'z'-axis labels
5+ x,y,z,conn_flag,value,annotate # see below

Key:

'x' x-axis value (float), NOT relative to left edge

'y' y-axis value (int 1..N), in same order as line #2 labels

'z' z/'color'-axis vaue (int)

'conn_flag'
this point is connected to the NEXT one (bool)

'value' numeric value (always 0 for lifelines), NOT scaled in any way

'annotate'
string of annotations, format is name='value'<spc>name='value'..

Example

```
XXX coming soon.
```

4.4.9 nldata_gp

Synopsis

```
Transform Lifelines, Load/Scatter (text) to Gnuplot command, data files
```

Usage

```
Usage: nldata_gp.py [-hcdi] gnuplot-cmd...
    -h           Print this message
    -c file      Command output file, defaults to stdout
    -d file      Data output file, defaults to inline with command file
    -i file      File to process, defaults to stdin
    -o file      Plot output file, by default just show on the screen
                 The gnuplot terminal type is inferred from the file
                 extension:
                      .gif  -> gif
                      .mif  -> mif (framemaker interchange format)
                      .png  -> png
                      .ps   -> postscript
    -v           Verbose output to stderr, e.g. show progress
    gnuplot-cmd  Commands passed through to gnuplot
```

Detailed description

nldata_gp.py transforms the output of nldata.py into input for gnuplot. By default, both the "command" and "data" parts of the gnuplot intput are written to standard output. The user can redirect either of these to a file.

Because they read from stdin and write to stdout by default, *nldata.py* and *nldata_gp.py* can be set up in a Unix pipeline to gnuplot itself. See [nldata_gp_examples], page 34.

Examples

- Display a gnuplot on the terminal from 'my.log', configuring how to graph it from 'my.cfg'.

  $ cat my.log | nldata.py my.cfg | nldata_gp.py | gnuplot -persist

- Write a gnuplot PNG file from 'my.log', configuring how to graph it from 'my.cfg'.

  $ cat my.log | nldata.py my.cfg | nldata_gp.py -o my.png | gnuplot -persist

4.4.10 nlprof

Synopsis

```
Provide a summary of timings in a logfile.
```

Usage

```
usage: nlprof.py [-ahp] -cDELIM -sSTART -eEND -kKEYS -iID_FIELD file

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -p, --prefix          Expect start.<something> (default=<something>.start)
  -cDELIM, --delimiter=DELIM
                        Set delimiter between <something> and the 'start' or
                        'end' string (default='.')
  -sSTART, --start=START
                        Set string that marks the start of a timed code block
                        (default='start')
  -eEND, --end=END      Set string that marks the end of a timed code block
                        (default='end')
  -kKEYS, --sort-keys=KEYS
                        Set sort keys. Results will be sorted by each key in
                        turn. Keys are (n)ame, (a)vg, (m)in, (M)ax, (c)ount,
                        (i)d(default='n'); Example: -k anc
  -a, --show-all        Show all timings in report
  -iID_FIELD, --identifier=ID_FIELD
                        [repeatable] Field name(s) to combine as an identifier
                        for separating start/end events with the same name
```

```
                        (default=None)
  -rFIELD, --rate=FIELD
                        Field name to use for calculating rate per second. The
                        formula applied is float(value) / elapsed_time
  -R, --csv             If TRUE, output comma-separated-values insteadof
                        aligned columns, and skip some header information
```

Detailed description

nlprof shows, in a simple but useful way, where time is being spent in your application. To provide the necessary raw data, you must first instrument your application to surround "performance-critical sections" with a pair of like-named events. You can do this by hand, or using the autolog facility. See [nlprof_example], page 35.

Example

Using nlprof really consists of 3 steps.

1. Instrument your code

```
import sys, nllite.nllite
g_log = nllite.nllite.LogOutputStream(sys.stderr)
def my_function(args):
    g_log.write("my_function.start",{"other":1,"data":2})
    # .. body of my_function
    g_log.write("my_function.end",{"whatever":"else"})
my_function(1)
```

2. Save sample output (Start/End events)

```
t DATE: 2004-07-07T06:32:14.252238
s LVL: INFO
i other: 1
i data: 2
s EVNT: my_function.start

t DATE: 2004-07-07T06:32:14.252601
s LVL: INFO
s whatever: else
s EVNT: my_function.end
```

3. Run *nlprof*

   Then, to "profile" these results (let's say they are in a log file called "my_function.log"), you would type:

```
$ python nlprof.py my_function.log
Timing report printed at 2004-07-07T06:33:51.303811
All times are in seconds
Distinct start/end blocks: 1
==========
Sort key: block name
```

```
Name                Avg       Min       Max       Count
----                ---       ---       ---       -----
my_function         0.000319  0.000319  0.000319  1
==========
End of report
```

4.5 Analysis

This section describes the analysis tools in the NetLogger toolkit.

4.5.1 nlcpu

Synopsis

```
Write a single message to a NetLogger output URL
```

Usage

```
usage: %nlcpu.py [ options ]


options:
  --version           show program's version number and exit
  -h, --help          show this help message and exit
  -a, --all           Report all types of utilization (default)
  -cNUM, --count=NUM  Repeat count times (default=1)
  -i, --idle          Report 'idle' utilization
  -s, --sys           Report 'system' utilization
  -u, --user          Report 'user' utilization
  -wSEC, --wait=SEC   Pause wait seconds between each display. (default=1)
```

Detailed description

Extract CPU information from the 'vmstat' program, and write it as NetLogger records.

4.5.2 nlganglia

Synopsis

```
Read Ganglia in, write NetLogger out.
```

Usage

```
usage: nlganglia.py [options] host [port(8649)]

options:
```

```
--version             show program's version number and exit
-h, --help            show this help message and exit
-iSEC, --interval=SEC
                      poll interval in seconds (60)
-oURL, --output=URL   Output URL. Repeatable (stdout)
-wSEC, --duration=SEC
                      runtime in seconds, 0=forever (0)

Metrics, default is --standard:
  -a, --all           Show all metrics available
  -s, --standard      Show only these metrics: ('load_one', 'load_five',
                      'cpu_user', 'cpu_system', 'cpu_idle', 'bytes_in',
                      'bytes_out', 'mem_free')
  -v, --verbose       Verbose output
```

Detailed description

Poll a Ganglia meta-daemon (gmetad) for information, then convert the information to NetLogger format and send it to a NetLogger destination (file, URL, etc.).

4.5.3 nlfindmissing

Synopsis

```
Find missing events in lifelines and generate new events
describing them.
```

Usage

```
usage: nlfindmissing.py [options] config-file < data-file

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -d, --debug           Print debugging info
  -v, --verbose         Print debugging info
  -r, --report-ids      Report ids, only, of anomalous lifelines
  -oURL, --output=URL   NetLogger output URL (stdout=='-')
  -p, --progress        Report progress to stdout
  -s, --streaming       Do not stop on input EOF
```

Detailed description

Look for missing events in a stream of NetLogger lifelines, and write these events out along with optional context information.

Configuration

Most of the behavior of this program is controlled by its configuration file. This file is in either in XML or CXML (See Section 4.3 [Configuration file formats], page 16), in a format shared by other tools such as Section 4.5.4 [show_histogram], page 40.

CXML version:

```
#CXML#
# Elements common to all analysis component configurations
config:
   # Missing lifeline configuration
   lifeline:
       # Required. Ordered list of events in lifeline
       events: [name,name,..]
       # Optional. Ordered list of event labels
       labels: [label,label,..]
       # Required. Set of event grouping fields
       groups: [group,group,..]
       # Required. Set of event id fields
       ids: [id,id,..]
       # Optional. How to 'time out' an incomplete lifeline.
       timeout:
           # Default=0. Minimum amount of time to wait.
           min: 0.. seconds
           # Default=-1 (Inf). Maximum amount of time to wait
           max: -1,min.. seconds
           # For dynamic timeouts, how many lifelines to
           # use as a baseline for a 'normal' completion time
           baseline: 0..  lifelines
           # Only 1 of the next two may be present:
           # For dynamic timeouts, what percentile of lifeline
           # completion times that a lifeline must exceeed before
           # timing it out.
           percentile: 0..100
           # For dynamic timeouts, how many standard deviations
           # a lifeline's time must exceed before timing it out.
           stddev: 0.. standard deviations
       # Required. Which missing events trigger an anomaly.
       missing:
           # any event in the lifeline
           any:
           # last event
           last:
           # list of events
           include: [event,event,..]
       # Optional. How much context to include with 'missing'
       # events. Default is count: 10.
       context:
```

```
            # number of lifelines of context
            count: number
            # number of standard deviations, in time, of context
            stdev: number-of-stdevs
            # Output all lines from input (requires 'mark' below).
            all:
        # Optional. Add a field NL.ANOM to all events,
        # set to 1 for anomalies and 0 otherwise.
        mark:
```

XML version:

```
<?xml version='1.0'?>
<!--  Elements common to all analysis component configurations -->
<config>
  <!--  Missing lifeline configuration -->
  <lifeline>
    <!--  Required. Ordered list of events in lifeline -->
    <events>
      <item>name</item>
      <item>name</item>
      <item>..</item>
      <!--  Optional. Ordered list of event labels -->
    </events>
    <labels>
      <item>label</item>
      <item>label</item>
      <item>..</item>
      <!--  Required. Set of event grouping fields -->
    </labels>
    <groups>
      <item>group</item>
      <item>group</item>
      <item>..</item>
      <!--  Required. Set of event id fields -->
    </groups>
    <ids>
      <item>id</item>
      <item>id</item>
      <item>..</item>
      <!--  Optional. How to 'time out' an incomplete lifeline. -->
    </ids>
    <timeout>
      <!--  Default=0. Minimum amount of time to wait. -->
      <min>0.. seconds
      <!--  Default=-1 (Inf). Maximum amount of time to wait --></min>
      <max>-1,min.. seconds
```

```
        <!--  For dynamic timeouts, how many lifelines to -->
        <!--  use as a baseline for a 'normal' completion time --></max>
        <baseline>0.. lifelines
        <!--  Only 1 of the next two may be present: -->
        <!--  For dynamic timeouts, what percentile of lifeline -->
        <!--  completion times that a lifeline must exceeed before -->
        <!--  timing it out. --></baseline>
        <percentile>0..100
        <!--  For dynamic timeouts, how many standard deviations -->
        <!--  a lifeline's time must exceed before timing it out. --></percentile>
        <stddev>0.. standard deviations
        <!--  Required. Which missing events trigger an anomaly. --></stddev>
      </timeout>
      <missing>
        <!--  any event in the lifeline -->
        <any>
          <!--  last event -->
        </any>
        <last>
          <!--  list of events -->
        </last>
        <include>
          <item>event</item>
          <item>event</item>
          <item>..</item>
          <!--  Optional. How much context to include with 'missing' -->
          <!--  events. Default is count: 10. -->
        </include>
      </missing>
      <context>
        <!--  number of lifelines of context -->
        <count>number
        <!--  number of standard deviations, in time, of context --></count>
        <stdev>number-of-stdevs
        <!--  Output all lines from input (requires 'mark' below). --></stdev>
        <all>
          <!--  Optional. Add a field NL.ANOM to all events, -->
          <!--  set to 1 for anomalies and 0 otherwise. -->
        </all>
      </context>
      <mark></mark>
    </lifeline>
  </config>
```

4.5.4 show_histogram

Synopsis

```
Show a histogram of lifeline latencies
```

Usage

```
usage: show_histogram.py [options] config-file < data-file

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -d, --debug           Print debugging info
  -fFORMAT, --format=FORMAT
                        Output format: ('table', 'text')
  -oOFILE, --output=OFILE
                        Output file (default=stdout)
  -p, --progress        Report progress to stdout
```

Detailed description

Look for missing events in a stream of NetLogger lifelines, and write these events out along with optional context information.

Index